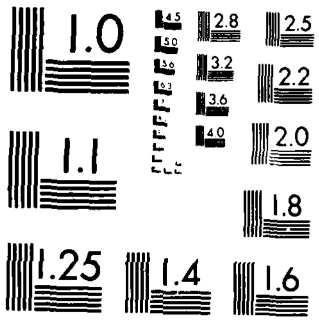
AD-A143 529 SIMULATION OF LARGE NETWORKS OF PROCESSORS BY SMALLER ONES(U) MARYLAND UNIV COLLEGE PARK CENTER FOR AUTOMATION RESEARCH S K BHASKAR ET AL. MAY 84 1/1

UNCLASSIFIED CAR-TR-63 AFOSR-TR-84-0558 F49620-83-C-0082 F/G 12/1 NL

END
FILMED
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

C4

CAR-TR-63
CS-TR-1401

F49620-83-C-0082
May 1984

SIMULATION OF LARGE NETWORKS OF PROCESSORS BY SMALLER ONES

S.K. Bhaskar
Azriel Rosenfeld

Center for Automation Research
University of Maryland
College Park, MD 20742

Angela Y. Wu

Dept. of Statistics and Computer Science
American University
Washington, DC 20016

COMPUTER VISION LABORATORY

CENTER FOR AUTOMATION RESEARCH

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

A 143 529

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 84-0558
6a. NAME OF PERFORMING ORGANIZATION University of Maryland	6b. OFFICE SYMBOL (If applicable)
7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research	
6c. ADDRESS (City, State and ZIP Code) Center for Automation Research College Park MD 20742	7b. ADDRESS (City, State and ZIP Code) Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR	8b. OFFICE SYMBOL (If applicable) NM
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-83-C-0082	
8c. ADDRESS (City, State and ZIP Code) Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO
61102F	2304	A2	

11. TITLE (Include Security Classification)
SIMULATION OF LARGE NETWORKS OF PROCESSORS BY SMALLER ONES.

12. PERSONAL AUTHOR(S)
S.K. Bhaskar, A. Rosenfeld, and A.Y. Wu

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	MAY 84	24

16. SUPPLEMENTARY NOTATION

COSATI CODES		
FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Parallel processing; multiprocessing; processor networks; interprocessor communication.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper considers the problem of simulating a large network N of processors using a small set of p processors. The approach taken is to partition the nodes of N into p subsets $N_1, \ldots, N_p$ and to assign each subset to a processor for simulation. In order to equalize the work-loads of the processors, the sizes of $N_1, \ldots, N_p$ should be as equal as possible; and in order to minimize (and equalize) the amount of message passing between the processors, the number of pairs of nodes that are neighbors in N but belong to different subsets should be [illegible] (and as equal) as possible. The authors discuss the general problem of partitioning a graph N so as to satisfy these criteria, and also consider the particular case of partitioning a tree.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Robert N. Buchal	22b. TELEPHONE NUMBER (Include Area Code) (202) 767- [illegible]
	22c. OFFICE SYMBOL NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

84 07 24 048

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

CAR-TR-63 F49620-83-C-0082
CS-TR-1401 May 1984

SIMULATION OF LARGE NETWORKS OF PROCESSORS BY SMALLER ONES

S.K. Bhaskar
Azriel Rosenfeld

Center for Automation Research
University of Maryland
College Park, MD 20742

Angela Y. Wu

Dept. of Statistics and Computer Science
American University
Washington, DC 20016

N sub 1 N sub P

ABSTRACT

This paper considers the problem of simulating a large network N of processors using a small set of p processors. The approach taken is to partition the nodes of N into p subsets $N_1, \ldots, N_p$ and to assign each subset to a processor for simulation. In order to equalize the workloads of the processors, the sizes of $N_1, \ldots, N_p$ should be as equal as possible; and in order to minimize (and equalize) the amount of message passing between the processors, the number of pairs of nodes that are neighbors in N but belong to different subsets should be as small (and as equal) as possible. We discuss the general problem of partitioning a graph N so as to satisfy these criteria, and also consider the particular case of partitioning a tree.

Research sponsored by the Air Force Office of Scientific Research (AFSC), under Contract F49620-83-C-0082. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The help of Janet Salzman in typing this report is gratefully acknowledged.

1. Introduction

There is a growing interest in the study (and eventually, the development) of large networks of processors for application to various computational tasks [1]. For example, it has long been recognized [2] that image processing can be done very efficiently on a mesh-connected network of the same size as the image - i.e., for an n by n digital image one should use an n by n processor array, one processor per pixel.

Cost considerations limit the size of the networks that can be constructed in practice. For example, the largest mesh-connected network yet constructed, NASA's Massively Parallel Processor [3], is an array of 128 by 128 processors; but most images are larger than this, typically 512 by 512 or more.

This paper addresses the problem of simulating a large network N of processors when only a small number p of processors are actually available. We shall assume, for simplicity, that the available processors can be connected in any desired way. Intuitively, it seems reasonable to do the simulation by partitioning the nodes of N into p sets $N_1,\ldots,N_p$, and let each processor simulate one of the sets. In Section 2 we shall define criteria that such a partition should satisfy in order to minimize the computation time required for the simulation. Section 3 considers algorithms for partitioning an arbitrary graph based on such criteria, Section 4 considers the special case of partitioning a tree, and Section 5 gives some further special results for the case of a complete binary tree.

2. Graph partitioning

The general graph partitioning problem can be formulated as follows: We are given a graph G (with node set N) and a set of p processors, which we can interconnect in any desired way. We want to partition N into p subsets $N_1,\ldots,N_p$, and assign each set to one of the processors. We must also interconnect the processors so that each of them can easily obtain information from the neighbors of its nodes in G.

In defining the partition, we want to make the amounts of computation done by the processors as equal as possible, in order to minimize the total computation time (if they were very unequal, all the processors would have to wait for the slowest one). Since each processor must simulate its nodes one at a time, this suggests that the *numbers of nodes assigned* to the processors should be as equal as possible. (This ignores the fact that the simulation may not take the same time for all nodes; for example, it may take longer for a node of high degree than for a node of low degree, since a high-degree node has more inputs. To take this into account, one could require that the <u>sums of the degrees</u> of the nodes, rather than the number of nodes, assigned to each processor to be as equal as possible. For simplicity, however, we shall use simple equalization of the numbers of nodes in our examples.)

We also want to minimize interprocessor communication, and to equalize the amounts of information that a processor may need to receive from the other processors in order to do the simulation, since a processor can receive only one message at a time. Let us assume, for simplicity, that to simulate a node of G we only need information from the neighbors (in G) of that node. Then, we would like to define the partition so as to minimize the number of neighbors (in G) of the nodes in each set N_i that lie in other sets N_j, and to make these numbers as equal as possible. Moreover, to avoid the need for relaying information through several processors (which might lead to queueing delays), we shall assume that processors i and j are directly connected if any node in N_i is a neighbor of any node in N_j. Thus the resulting processor network G' is a <u>condensation</u> of the graph G; each node i of G' corresponds to a set N_i of nodes of G, and nodes i,j of G' are joined by an arc iff some node in N_i was joined by an arc of G to some node in N_j.

Before discussing how to construct such condensations for arbitrary graphs, let us briefly consider the case where G is a mesh-connected array. Here, to minimize the number of neighbors of N_i that lie in N_j for each pair of node sets $(i \neq j)$, we want the N_i's to be "compact" blocks of nodes that contain as few border nodes (=nodes with neighbors not in N_i) as possible. It is not hard to see [4] that square blocks of nodes are best for this purpose - i.e., they have the fewest border nodes for a given area. Thus to efficiently simulate a large mesh-connected array, we should condense it by dividing it into square blocks of equal size; note that such a condensation is itself a mesh-connected array. Incidentally, it makes little difference whether we equalize the numbers of nodes in the blocks or the sum of the degrees of these nodes, since most of the nodes in an array (i.e., all but the nodes on its border) have the same degree.

A1

3. Arbitrary graphs

The problem of graph condensation so as to minimize a given cost function was solved in a general setting in [5]. The method used was to apply dynamic programming to solve a recursion equation expressing the changes in cost variables arising from the addition of a single node to one of the previously computed partitions. The complexity of the solution grows asymptotically as x^x where x is a function of the degree of the graph. However, in the particular case of partitioning trees, a linear time solution (linear in the number of nodes in the tree) was obtained. It is therefore of interest to investigate whether there exist heuristic methods which produce acceptable solutions.

In this section, we first consider a method of graph partitioning designed to minimize the number of "outlinks" between the subsets, i.e. the number of node pairs (a,b) such that (a,b) is an arc of G but a and b belong to different subsets of the partition. Our approach is based on the same principles as Huffman coding [6], but using the degrees of the vertices. The idea is to form groups around vertices of large degrees, repeatedly "merging" other vertices together until the number of vertices remaining equals the number of host vertices. The criterion used to pick two vertices to be merged is derived informally as follows:

Suppose that A is a vertex with large degree n, and suppose we merge A with some partition N_i (of which A is a neighbor). In so doing, we are adding n-1 to the number of inputs that the processor handling N_i must deal with. Thus, whenever we merge, we should merge vertices of the least possible degrees. Also, to preserve connectedness inside components we insist that the vertices to be merged be adjacent.

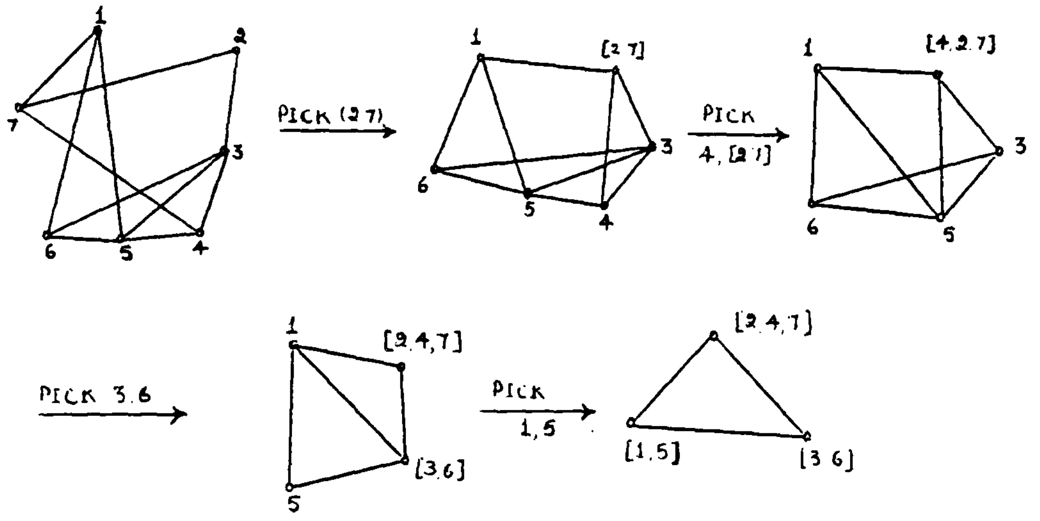
The definition of a merge between two vertices of a graph G is as follows:

```
Function merge (G,u,v);
(* returns a graph in which vertices u and v of G are merged *)
begin
    $E' \leftarrow \{(w,u) \mid (w,u) \in E(G)\} \cup \{(w,v) \mid (w,v) \in E(G)\}$
    $V' \leftarrow \{w \mid (w,u) \in E(G)\} \cup \{w \mid (w,v) \in E(G)\}$
    $E'' \leftarrow \{([uv],w) \mid w \in V'\}$
    return $(V(G) - \{u,v\} + \{[uv]\}, E(G) - E' + E'')$
end
```

The partitioning algorithm based on merging pairs of least degree is then as follows:

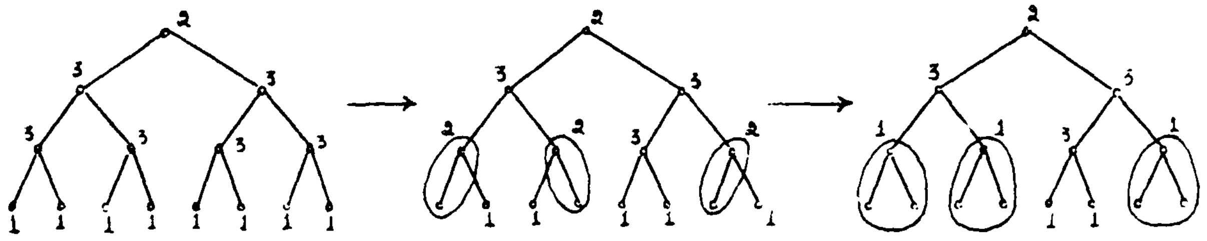
```
begin   {p = # processors available; G=(V,E) is the input graph}
    $G' \leftarrow G$;
    while $|V(G')| > p$ do
      begin  find the lexicographically smallest pair (deg(u),
             deg(v)) where $uv \in E(G)$;
          $G' \leftarrow$ merge (G',u,v)
      end
end.
```

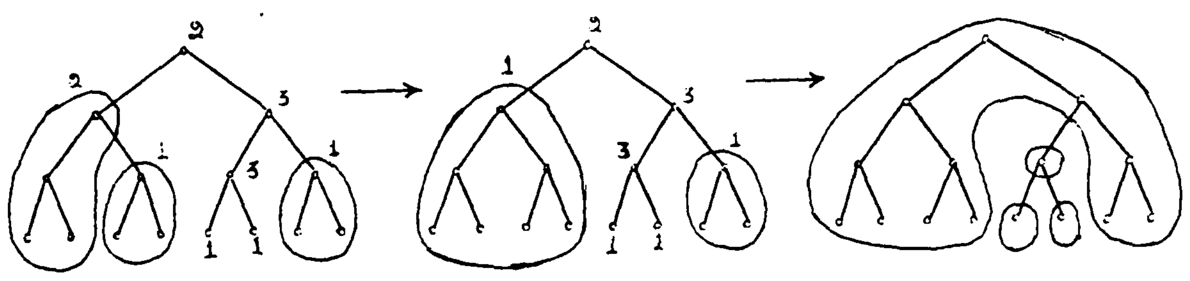
The partition found in this way is not necessarily unique if ties are resolved arbitrarily during the search for the smallest pair (deg(u), deg(v)). The following is an example of applying the partitioning algorithm to a graph G for p=3:



Thus the final partition is [2,4,7],[1,3],[3,6]. In this partition, the nodes of [2,4,7] have 4 outlinks; those of [1,5] have 5; and those of [3,6] have 5. Thus the numbers of outlinks have been (approximately) equalized.

A defect of this algorithm is that it does not require the sets N_i to contain approximately equal numbers of nodes. For example, if p=4 and G is a complete binary tree of height 3, the following partition can be obtained:





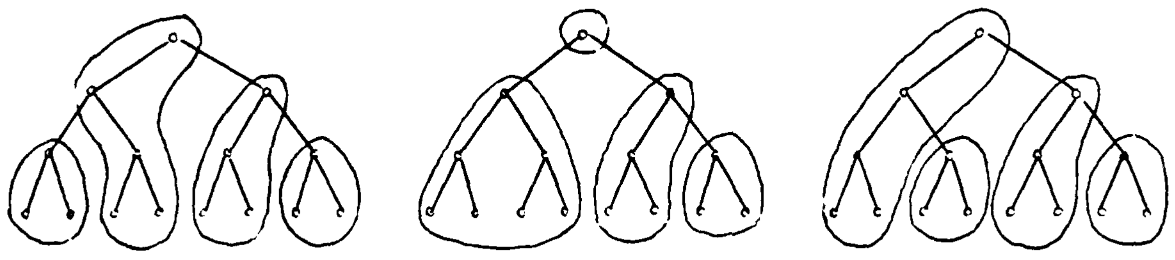
To obtain a more equitable distribution of nodes, we must also take into account the number of nodes in the sets N_i. We can do this by using a linear combination of weight (=number of nodes in N_i) and degree (of the node N_i) as the criterion for selecting nodes to be merged.

Thus, for each node v, we have

$$f(v) = \alpha.\text{weight}(v) + \beta.\text{degree}(v)$$

and we can pick the constants α and β to "tune" the rtition.

For the same tree considered above, the followir parti- tions, in all of which the penalty is 2, can be obta .r .:



$\alpha = \beta = 1 \quad \rho = 4$ $\qquad \alpha = 1,\ \beta = 3,\ \rho = 4$ $\qquad \alpha = 3 \text{ or } 5,\ \beta = 1,\ \rho = 4$

It should be emphasized that the partitions obtained in this way are dependent on the order in which nodes with equal f-values are chosen. Thus, for highly symmetric structures, wildly varying partitions can be obtained.

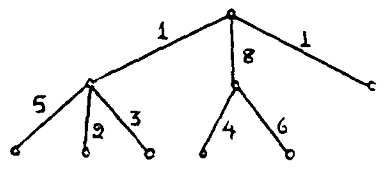
The time complexity of the algorithm is determined as follows: Assume that the graph has n nodes. Thus there are n f-values and the least is found in $O(n)$ time. The loop is executed $(n-p)$ times (assuming $n \geq p$) giving an overall complexity of $(n-p)\ O(n)$ which is $O(n^2)$ for fixed p.

It has been found empirically that having $\beta > \alpha$ usually results in components with very uneven distributions of nodes per component. This is because it is possible for a component with a large number of nodes to have small degree (as in the second example above). Such a node will then become a candidate for further merging, adding to its already large size. Having $\beta = 0$ will merge nodes based on weight alone, without regard for the degrees of the nodes being merged. This will lead to large numbers of outlinks. Thus it is best to have $\alpha > \beta$, $\beta \neq 0$. It has also been found, empirically, that as long as $\alpha > \beta$, increasing the value of α does not have much effect. An intuitive explanation for this may be given as follows: Suppose that $\{v_1, v_2, \ldots, v_n\}$ is the set of nodes at some stage, and that v_i and v_j are selected for merging. This means that $(f(v_i), f(v_j))$ [or $(f(v_j), f(v_i))$] is the smallest, lexicographically - that is, $(\alpha.\text{weight}(v_i) + \beta.\deg(v_i),$ $\alpha.\text{weight}(v_j) + \beta.\deg(v_j))$ is the smallest. Since α is "large", the small f-value of v_i (or v_j) must be due to a "small" value of weight (v_i) (or weight (v_j)). If we now increase α to $\alpha + k$ $(k > 0)$, the set of f-values becomes $\{f(v_1) + k.\text{weight}(v_1), \ldots, f(v_n) + k.\text{weight}(v_n)\}$. If v_i and v_j were picked with the old value of α, because of the small weight of v_i the increase caused by adding $k.\text{weight}\ (v_i)$ is usually not enough to keep v_i and v_j from still having the smallest f-values.

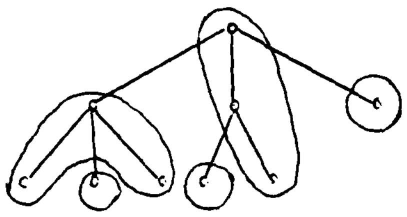
4. Trees

For trees, a solution to the partitioning problem is given in [5] which is linear in the number of nodes in the tree. Integer weights are attached to both nodes and arcs of the tree, and the solution finds a partition in which the weight of each component is less than or equal to a parameter, w, and the sum of the weights on the arcs that form outlinks is minimal.

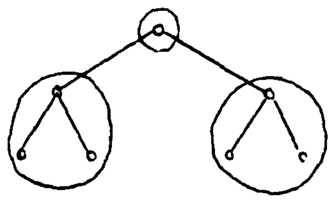
For example, consider the tree



Here, all nodes have unit weight, w=3, and the numbers alongside the arcs are their weights. In this case, the following optimal partition is obtained:



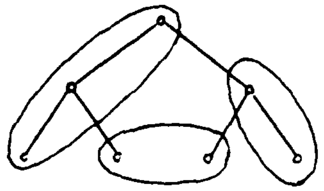
where we have 5 components, each with ≤3 nodes. The cost of the outlinks is 1+2+4+1=8, and this is the least possible value. However, the distribution of nodes per component is not even. For example, if we consider the tree



with unit weights on the arcs and nodes and with w=3, the solution is optimal in terms of outlinks, but has an uneven distribution of nodes. To achieve a more even distribution of nodes per component, the tree nodes must be weighted differently so that clusters are broken up. In practice, it is difficult to determine just what this weighting should be.

If we do not use weightings, then any partition of a tree into p connected components has the same number of outlinks, namely p-1. To see this, suppose that a tree G having n nodes is partitioned into p connected components; thus each component is itself a tree. If component N_i has n_i nodes, $1 \le i \le p$, we must have $\sum_{1 \le i \le p} n_i = n$. The total number of arcs internal to the components is $\sum_{1 \le i \le p} (n_i - 1) = n - p$. Since the total number of arcs in the tree G is n-1, there must be $(n-1)-(n-p)=p-1$ arcs not internal to any component. These are just the arcs connecting together the p components. Thus the total number of outlinks is p-1.

In many cases, it is not optimal to insist on connected components in a partition of a tree. For example, an optimal partition of the tree shown below into three components results in one of the components being disconnected. It can be verified that in any partition of this tree into subgraphs having 2,2, and 3 nodes, at least one of the subgraphs must be disconnected.



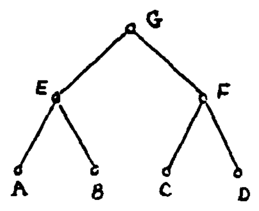
5. Complete binary trees

In this section we consider methods of partitioning complete binary trees. We recall that a complete binary tree has 2^m-1 nodes for some m: 1 root node, 2 nodes at the level below the root, 4 nodes at the level below that, ... , 2^k nodes k levels below the root, ... , and 2^{m-1} leaf nodes.

We saw in Section 4 that any partition of a tree into p connected parts has p-1 outlinks. Thus the figure of merit of such a partition depends only on how nearly equal are the numbers of nodes in the parts; the best partition would be one in which all the parts are of the same size. We now show that for a complete binary tree, there is only one way of partitioning it into connected parts that are all of the same size.

Proposition. Suppose that a complete binary tree is partitioned into p connected parts all of the same size. Then the parts are all complete binary trees, say of size 2^h-1, and their roots are at levels 0,h,2h,... below the root of the given tree. In particular, we must have $p = 1+2^h+2^{2h}+...2^{m-h}$; h divides m, and $2^m-1=p(2^h-1)$.

Proof: Let A be a leaf of the tree, e.g., as shown (partially) below:

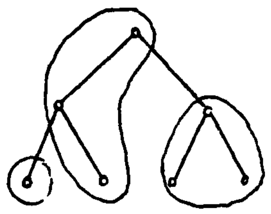


If {A} is a part, the parts are all singletons and the proposition is trivially true with h=1. If not, the part N_A containing A must also contain E, since E is the only node adjacent to A. But then N_A must also contain B, since otherwise B could not belong to a connected part having at least two nodes. If N_A = {A,B,E}, it is a complete binary tree having three nodes and the parts all contain three nodes. A similar argument now shows that the part containing C must be {C,D,F}, and similarly each pair of leaves having a common father must constitute a part. If we remove these parts, what remains is a complete binary tree of height m-2, and we can repeat the argument for that tree; we conclude ultimately that the parts are all complete binary trees having three nodes (i.e., h=2), and that m is even (i.e., h divides m).

Suppose next that N_A has more than three nodes; then it must contain G. But then {C,D,F} cannot belong to a connected part having more than three nodes; thus these nodes too must belong to N_A, so that it has at least seven nodes. If it has exactly seven, it is a complete binary tree having seven nodes {A,B,C,D,E,F,G}, and all the parts have seven nodes. An argument analogous to that in the previous paragraph then shows that all the parts are complete binary trees having seven nodes (i.e., h=3), and that m is a multiple of 3.

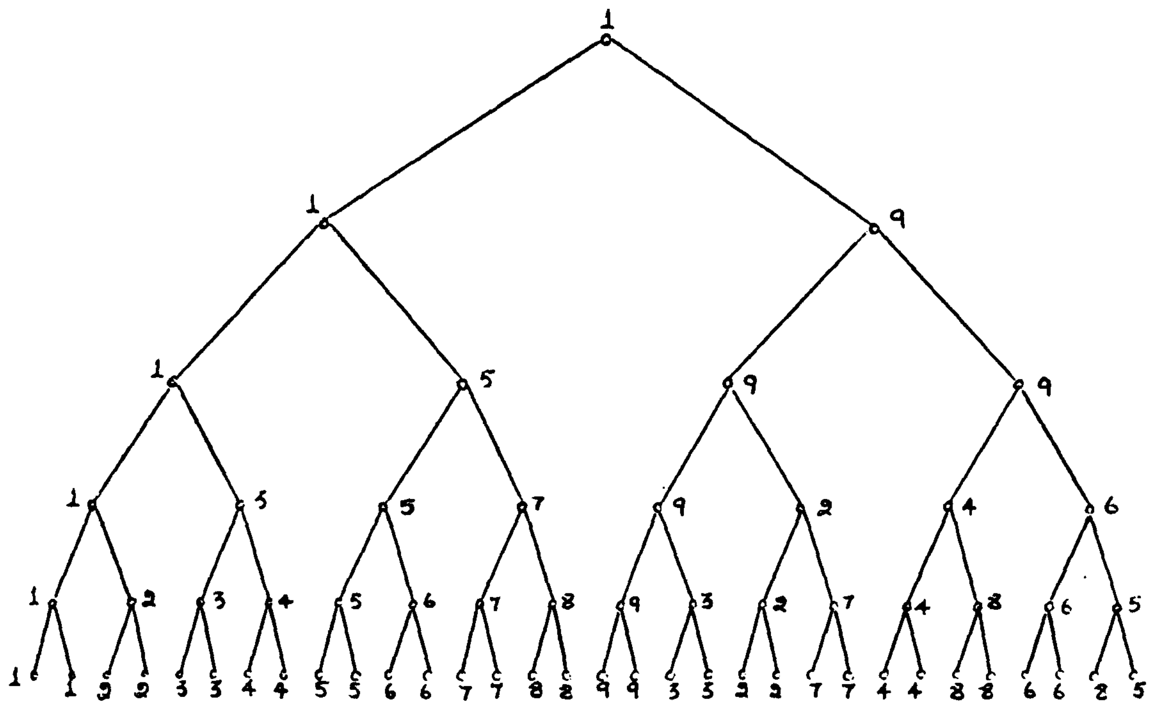
We can repeat this reasoning to show that if N_A has more than seven nodes, it has at least 15, in which case all the parts are complete binary trees having 15 nodes (h=4); if it has more than 15, it has at least 31; and so on. Thus the Proposition is true in any case. ||

Note that if we relax the condition in the proposition to allow just one part to have a different number of nodes than all the others, the parts need no longer be complete binary trees. For example, in



two parts have three nodes each and the third part has one, but one of the parts is not a complete binary tree. However, there does always exist a partition of a complete binary tree of size 2^m-1 into complete binary trees, all but (at most) one of which are of size 2^h-1, for any desired $h \leq m$. Indeed, let $m=kh+r$, where $0 \leq r \leq h$. Then the parts consist of: a tree of size 2^r-1 rooted at the root; 2^r trees of size 2^h-1 rooted r levels below the root; 2^{r+h} trees of size 2^h-1 rooted $r+h$ levels below the root; 2^{r+2h} trees of size 2^h-1 rooted $r+2h$ levels below the root; ...; $2^{r+(k-1)h}$ trees of size 2^h-1 rooted $r+(k-1)h$ levels below the root.

Unfortunately, partitions of a complete binary tree into connected parts are not always optimal even if the parts are all of exactly the same size. For example, as shown below, the complete binary tree with 63 nodes can be partitioned into 9 components of 7 nodes each, where the maximum number of outlinks from any component is 5 (arising from component no. 9; the nodes are labelled with their component numbers). On the other hand, if we partition this tree into 9 complete binary trees of 7 nodes each, the tree rooted at the root has 8 outlinks (to the 8 trees rooted 3 levels below the root); thus the partition into complete binary trees is not optimal.



A partition into subtrees is in some sense more "natural" than a partition in which the components are disconnected and dispersed far apart. This is because it natural for a large tree to be simulated by a smaller, similar tree. This also preserves the original tree structure within the components so that algorithms written for the simulation are more natural in expressing the flow present in the original larger tree.

If n and p are such that it is not possible to assign complete binary trees to the sets N_i, we can attempt to assign <u>sets</u> of complete trees to each component. For symmetry, we would like the complete subtrees in a set N_i to all be of the same size. This, combined with the fact that nothing is gained by placing the equal-sized subtrees at different levels (the number of outlinks cannot decrease), and that placing the complete subtrees "adjacent" to each other preserves the original tree structure within a component, leads us to consider forming each set N_i from equally sized complete subtrees at the same level. We shall refer to such N_i as "trapezoidal" blocks.

To completely specify the shape of a trapezoidal block, we must specify the number of nodes in its top edge and the number of its layers. It is possible that differently shaped trapezoidal blocks can have the same number of nodes. In such cases, the block with fewest outlinks should be picked.

Suppose that we have two trapezoidal blocks, one with m_1 nodes in its top edge and ℓ_1 layers deep, and the other with m_2 nodes in its top edge and ℓ_2 layers deep. Suppose further that both contain the same number of nodes. Thus $m_1(2^{\ell_1}-1) = m_2(2^{\ell_2}-1)$. Also, assume that $m_1>m_2$; thus $\ell_2>\ell_1$. The number of outlinks for the first block is $m_1(2^{\ell_1}+1)$and for the second it is $m_2(2^{\ell_2}+1)$.

Since $m_1>m_2$ we have

$$\begin{aligned} 2^{\ell_1}m_1+m_1 &> 2^{\ell_1}m_1 - m_1 + m_2 + m_2 \\ &> 2^{\ell_2}m_2 - m_2 + 2m_2 \\ &> 2^{\ell_2}m_2 + m_2 \end{aligned}$$

Hence the first trapezoidal block will have more outlinks than the second.

Thus, we should use trapezoids that have as many layers as possible, and tops as small as possible - in other words, they should consist of as few trees as possible.

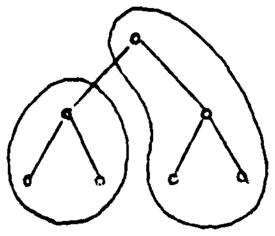
The following algorithm determines suitable parameters for the trapezoid:

```
begin   $i \leftarrow \lceil \frac{2^m-1}{p} \rceil$; $a \leftarrow 0$;
   while $i \neq (2^b-1)$ for some b do
      begin   $a \leftarrow a+1$; $i \leftarrow \lceil i/2 \rceil$
      end;
   if   $i = 2^b-1$ for some b then
      begin
         number of nodes in top edge = $2^a$;
         number of layers = b
      end
end
```

Clearly, the algorithm stops for the least possible value of a and the highest possible value of b. The condition $i=2^b-1$ for some b is easily detected from the binary representation of b, and is needed to obtain complete subtrees within each trapezoid. Note that if p divides 2^m-1 evenly, we have $i = (\frac{2^m-1}{p})p = 2^m-1$ and the *while* loop is not executed. We then have a=0, so that the trapezoidal blocks are complete binary subtrees.

The trapezoidal blocks obtained by this algorithm are not optimal in the number of outlinks. For example, if p=2, the following partition of a complete binary tree is obtained:

with a penalty of 4, whereas the optimal partition is



with a penalty of 1.

The following algorithm* partitions the n nodes of a complete binary tree T such that the nodes are distributed to the p processors as evenly as possible, and it gives an upper bound to the cost (maximum number of outlinks) of partitioning a complete binary tree.

Step 1. Find $x_i \in \{0,1\}$ such that $t = \frac{n}{\lfloor p \rfloor} = \sum_{i=0}^{k} x_i 2^i$ where $x_k \neq 0$. Clearly $k = \lfloor \log_2 t \rfloor$.

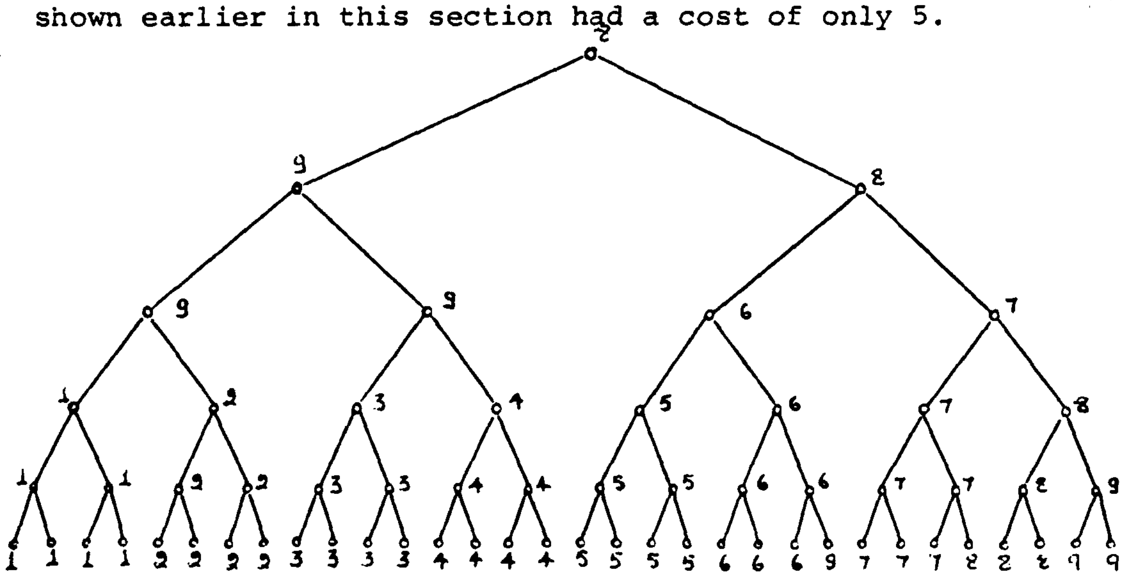
Step 2. To each processor, for each i, $1 \le i \le k$, such that $x_i = 1$, assign a complete subtree with $2^i - 1$ nodes such that its leaves are also leaves of the original tree T. There are enough leaves in T, because if $x_i = 1$, then we need 2^{i-1} leaves for each processor. Altogether $p \sum_{i=1}^{k} x_i 2^{i-1}$ leaves are needed. But $p \sum_{i=1}^{k} x_i 2^{i-1} = (p \sum_{i=1}^{k} x_i 2^i)/2 \le$ $pt/2 \le n/2 \le$ number of leaves in T.

*This algorithm is based in part on suggestions by Q. Stout.

Step 3. Let s = number of nonzero x_i's, $0 \le i \le k$ $(s \le k+1)$. Among the unassigned nodes of T, assign s nodes (in any convenient way) to each of the p processors. Now each processor has t nodes, and there are n-pt unassigned nodes.

Step 4. Choose (arbitrarily) n-pt processors to receive one more node each. In other words, n-pt processors have t+1 nodes and (t+1)p-n processors have t nodes.

Each of the complete subtrees in step 2 has one outlink and the number of such complete subtrees in each processor is $\le k$. Each node assigned in steps 3 and 4 introduces at most three outlinks. Thus, the total number of outlinks in each processor is $\le k+3(s+1) \le 4k+6 = 4\lfloor \log \frac{n}{p} \rfloor + 6$. In general, this cost can often be smaller if in the assignment of nodes in steps 3 and 4, we are careful in assigning neighboring nodes to the same processor whenever possible. However, this method of partitioning does not always result in optimal cost, as the following example shows; the cost of this partition is 7, arising from component 9, whereas the partition of this same tree shown earlier in this section had a cost of only 5.



The above algorithm gives an upper bound on the cost of partitioning a complete binary tree. The components obtained are very "nonuniform" in the sense that the components can be disconnected; the nodes in each component are connected into graphs of different shapes; and the nodes are all on different levels of the original tree. The resulting structure is also not a tree.

6. Concluding remarks

In order to simulate a large network N of processors using a smaller set of processors, a natural approach is to divide N into parts and let each processor simulate one of the parts. In order to equalize the workloads of the processors, the number of nodes of N in the parts should be as equal as possible, and the numbers of arcs between parts should be as small (and as equal) as possible. "Natural" partitions that are optimal with respect to these criteria can be found for certain special N's; for example, if N is a mesh-connected array, partitioning it into square blocks is optimal. For general graphs, finding an optimal partition has exponential cost. We suggest a suboptimal partitioning algorithm that has quadratic complexity. For trees, an algorithm of linear complexity can be given that minimizes the number of arcs between parts while keeping the number of nodes in each part bounded, but it is not obvious how to modify this algorithm to equalize the number of nodes in the parts. In any case, in an optimal partition of a tree, the parts in general are not connected subgraphs. We give a suboptimal algorithm for partitioning a complete binary tree, and also discuss a class of "natural" partitions of such a tree into complete binary trees or into "trapezoidal" sets of such trees.

References

1. L. Uhr, Algorithm-Structured Computer Arrays and Networks, Academic Press, New York, 1984.

2. S. H. Unger, A computer oriented toward spatial problems, Proc. IRE, 46, 1744-1750, 1958.

3. K. E. Batcher, Design of a massively parallel processor, IEEE Trans. Computers, 29, 836-840, 1980.

4. T. Kushner, A. Y. Wu and A. Rosenfeld, Image processing on ZMOB, IEEE Trans. Computers, 31, 943-951, 1982.

5. J. A. Lukes, Combinatorial solutions to partitioning problems, Technical Report No. 32, Stanford Electronics Laboratories, Stanford University, 1974.

6. D. E. Knuth, Fundamental Algorithms, Addison-Wesley, Reading, MA, 1973.